5. Rasterisierung

Um graphischen Output auf einem Gerät zu erzeugen, muss die verwendete Programmiersprache Befehle dafür zur Verfügung stellen. Die mit diesen Befehlen erzeugbaren einfachen Graphikbausteine nennt man auch *Graphik-primitive*. Solche Bausteine sind neben einfachen Zeichnungen auch Formatierungs-Anweisungen und Meta-Informationen. Die wichtigsten solchen Befehle sind:

In 2D:
- Punkte, Linien
- Polygone, Kreise, Ellipsen und andere Kurven, alles auch gefüllt
- Bitmap-Operationen
- Buchstaben und Zeichen

In 3D:
- Dreiecke und andere Polygone
- Freiformflächen

Darüber hinaus braucht man noch Befehle um die Eigenschaften der Primitive zu definieren, z.B. Farbe, Füllmuster, Textur, Materialeigenschaft, Transparenz. Diese Befehle bewirken meist, dass alle danach erzeugten Primitive die zuletzt definierten Eigenschaften annehmen.

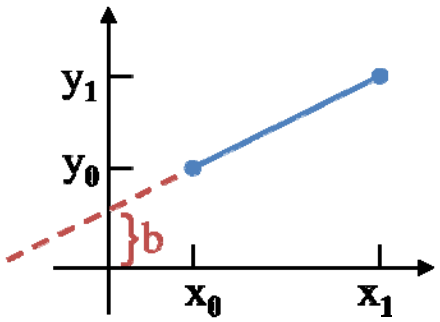
Linienalgorithmen

Insbesondere das Zeichnen gerader Linien auf Rastergeräten ist eine wichtige Operation. Das Basisverfahren DDA (Digital Differential Analyzer) wurde von Bresenham durch geschickte Umformung so gestaltet, dass es nur mit Integer-Operationen auskommt und damit *schneller* und *leichter in Hardware implementierbar* wurde.

Notation: eine Linie wird in der Form $y = mx + b$ angegeben,
wobei m den Anstieg der Linie beschreibt
und (0,b) der Schnittpunkt mit der y-Achse ist.

Aus den Endpunkten (x_0,y_0) und (x_1,y_1) der Linie lassen sich m und b berechnen:

$m = (y_1 - y_0) / (x_1 - x_0)$ $\qquad$ $b = y_0 - mx_0$



DDA-Verfahren

Der ***DDA-Algorithmus*** für $|m| < 1$ zählt zu y_0 für jeden Schritt nach rechts (x+=1) den Wert m dazu und rundet das Ergebnis danach auf ganze Zahlen. Dadurch entsteht eine Linie, bei der für jeden x-Wert genau ein Pixel für die Linie erzeugt wird.

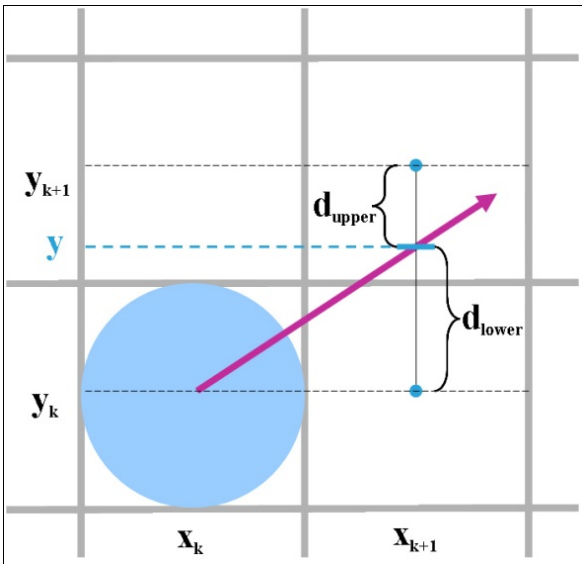
```
dx = x1 – x0; dy = y1 – y0;
m = dy / dx;

x = x0;  y = y0;
setPixel (round(x), round(y));

for (k = 0; k < dx; k++)
  { x += 1; y += m;
  setPixel (round(x), round(y)}
```

Für $|m| > 1$ werden x und y vertauscht, und das Verfahren wird in senkrechter Richtung durchgeführt. Auch der nachfolgende Bresenham-Algorithmus wird nur für $0 < m < 1$ dargestellt, die anderen Richtungen erhält man durch Spiegelung und durch Rotation um 90°.

Bresenham-Verfahren

Der ***Bresenham-Algorithmus*** erzeugt exakt dasselbe Ergebnis wie der einfache DDA, verwendet jedoch nur Integer-Arithmetik. Er ist dadurch schneller, leichter in Firm- oder Hardware zu implementieren, und überdies lässt er sich auch einfach für andere Kurven anpassen, z.B. Kreise, Ellipsen, Spline-Kurven usw.

Für $0 < |m| < 1$ wird ausgehend von der bekannten Lage des Pixels in der Spalte x_k für x_{k+1} nicht der exakte y-Wert berechnet, sondern lediglich eine Entscheidung getroffen, ob y_k oder y_{k+1} näher zum exakten y-Wert liegen.



Aus $y = mx + b$ folgt für die Spalte rechts von x_k der exakte y-Wert

$$y = m\cdot(x_k + 1) + b$$

Der Abstand zu y_k ist $\mathbf{d_{lower}} = y - y_k = m(x_k + 1) + b - y_k$
der Abstand zu y_{k+1} ist $\mathbf{d_{upper}} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

Wenn nun die Differenz $d_{lower} - d_{upper} = 2m\cdot(x_k + 1) - 2y_k + 2b - 1$ negativ ist, dann wird der untere Punkt (x_{k+1},y_k) gewählt, wenn sie positiv ist wird (x_{k+1},y_{k+1}) gewählt.

Setzt man für $m = \Delta y/\Delta x$ ein ($\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$) , und multipliziert diese Differenz mit Δx so erhält man eine Entscheidungsvariable:

$$p_k = \Delta x\cdot(d_{lower} - d_{upper}) = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c ,$$

die *das gleiche Vorzeichen* wie $d_{lower} - d_{upper}$ hat, aber keine Division erfordert.

Nun kann man ganz leicht aus der Entscheidungsvariablen für x_k: $p_k = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$ die Entscheidungsvariable für x_{k+1} berechnen:

$$\mathbf{p_{k+1} = 2\Delta y\cdot x_{k+1} - 2\Delta x\cdot y_{k+1} + c + p_k - 2\Delta y\cdot x_k + 2\Delta x\cdot y_k - c = p_k + 2\Delta y - 2\Delta x\cdot(y_{k+1} - y_k)}$$

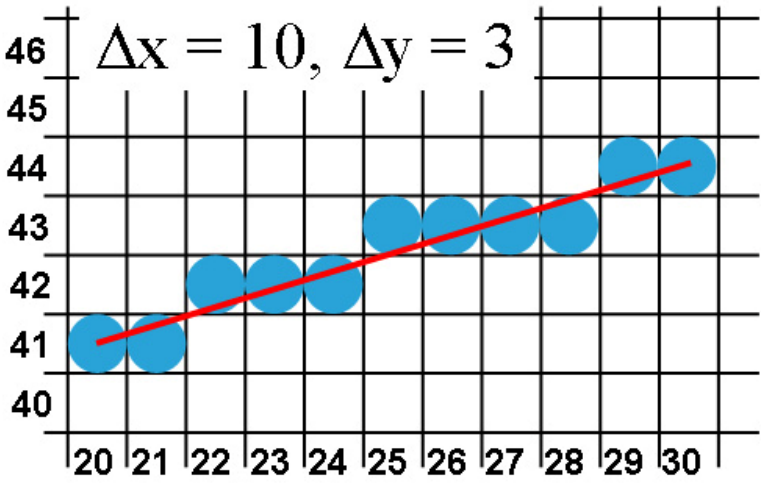
also lediglich durch Addition einer Zahl, die für alle Punkte der Linie konstant bleibt.
Als Anfangswert verwendet man natürlich $p_0 = 2\Delta y - \Delta x$.

Damit sieht der Bresenham-Algorithmus etwa so aus:

```
1. store left line endpoint in (x0,y0)
2. plot pixel (x0,y0)
3. calculate constants Δx, Δy, 2Δy, 2Δy - 2Δx,  and obtain p0 = 2Δy - Δx
4. At each xk along the line, perform test:
      if pk<0
      then plot pixel (xk+1,yk);  pk+1 = pk+ 2Δy
     else plot pixel (xk+1,yk+1);  pk+1= pk+ 2Δy - 2Δx
5. perform step 4 (Δx – 1) times.
```

Beispiel:

k	p_k	(x_{k+1},y_{k+1})
		(20,41)
0	-4	(21,41)
1	2	(22,42)
2	-12	(23,42)
3	-6	(24,42)
4	0	(25,43)
5	-14	(26,43)
6	-8	(27,43)
7	-2	(28,43)
8	4	(29,44)
9	-10	(30,44)

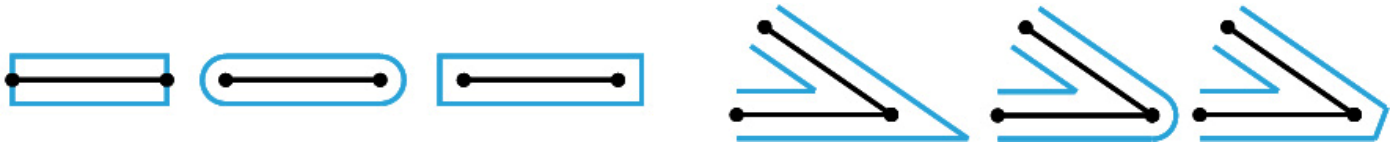


Attribute

Graphikprimitive können mit vielerlei Eigenschaften erzeugt werden, sogenannten *Attributen*.

Attribute von (Punkten und) Linien

Neben allgemein bekannten Eigenschaften von Linien, wie Strichdicke, Strichlierungsmuster, Farbe oder Pinseltyp, gibt es noch ein paar Attribute, die einem oft weniger bewusst sind. Dazu gehören etwa die Linienenden bei breiteren Linien sowie die Form von Ecken bei breiten Linien:



Weiters ist Antialiasing auch für Linien ein Thema, dazu werden etwas weiter unten Details gebracht.

Attribute von Text

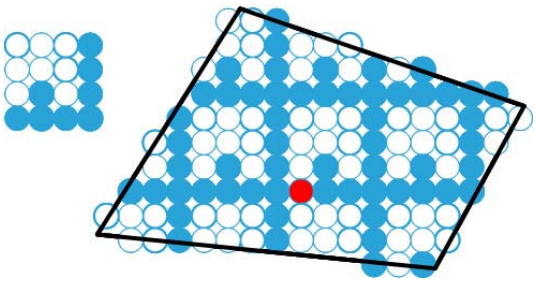
Die Eigenschaften, die Text annehmen kann, sind heute weitgehend Allgemeinwissen: Font (z.B. Courier, Arial, Times, **Broadway**, …), Stil (normal, **fett**, *kursiv*, unterstrichen, …), Größe, Richtung, Farbe, Bündigkeit (links, rechts, mittig, Blocksatz) und so weiter.

Sfzrn
Sfzrn

Fonts mit Serifen (oben) eignen sich besser für Fließschrift, Fonts ohne Serifen für plakativen Text. Die Repräsentation von Fonts erfolgt normalerweise durch die Definition der Umrisskurven der Buchstaben, für manche Anwendungen auch durch Pixelraster.

Attribute von (2D-) Polygonen und Flächen

Klarerweise sind die Attribute des Randes von Flächen dieselben wie die von Linien. Dazu kommt nun die Fläche selbst, die mit einer Füllung versehen werden kann. Muster werden dabei gewöhnlich durch repetitive Aneinanderreihung eines Grundmusters ausgehend von einem Referenzpunkt (auch Seed-Point genannt) erzeugt.



In vielen Anwendungen ist es auch notwendig, eine Kombination des neu gezeichneten Musters mit dem Hintergrund zu erzeugen. Hier gibt es viele Varianten, die oft auf logischen Verknüpfungen aufbauen: AND, OR, XOR. Das Mischen von Farben erfolgt meist durch Linearkombination der vorhandenen Hintergrundfarbe B mit der zu zeichnenden Vordergrundfarbe F: $P = t \cdot F + (1-t) \cdot B$

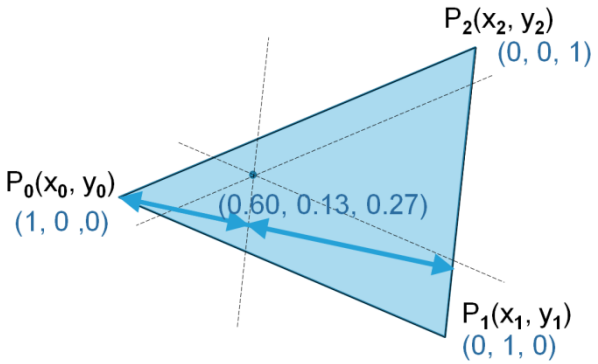
Baryzentrische Koordinaten

Baryzentrische Koordinaten sind eine Grundlage für die Interpolation von Pixeln in Dreiecken.

Dreiecke rasterisieren

Um Dreiecke zu füllen verwendet man oft *baryzentrische Koordinaten*. Jeder Punkt der Ebene wird dabei als gewichtetes Mittel der drei Eckpunkte des Dreiecks dargestellt:

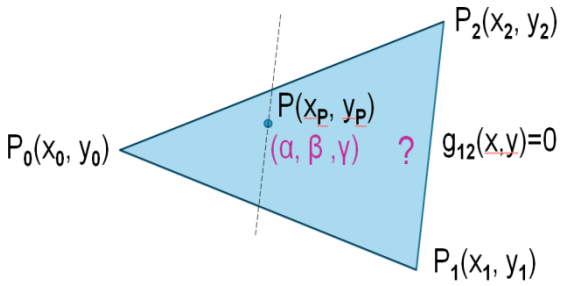
$$P=\alpha P_0 + \beta P_1 + \gamma P_2.$$



(α, β, γ) nennt man dann die baryzentrischen Koordinaten des Punktes P, wobei immer gilt: $\alpha+\beta+\gamma=1$. Alle Punkte mit $(0<\alpha<1,\ 0<\beta<1,\ 0<\gamma<1)$ liegen innerhalb des Dreiecks; sobald einer dieser Werte negativ oder größer als 1 ist, liegt der Punkt außerhalb des Dreiecks.

Zum Füllen eines Dreieckes berechnet man für jedes Pixel einer (möglichst engen) Umgebung dessen baryzentrische Koordinaten und zeichnet alle für deren Mittelpunkt $(0<\alpha<1,\ 0<\beta<1,\ 0<\gamma<1)$ gilt. Dabei kann man sehr einfach beliebige Eckpunktattribute (z.B. Farbe) in jedem Pixel mit (α, β, γ) gewichtet berechnen, dies entspricht einer linearen Interpolation dieser Werte.

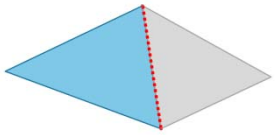
Berechnung der baryzentrischen Koordinaten



Sei $g_{12}(x,y) = a_{12}x+b_{12}y+c_{12}=0$ die Trägergerade durch die Punkte P_1 und P_2, dann berechnet sich α des Punktes $P(x_P,y_P)$ zu

$$\alpha = g_{12}(x_P,y_P) / g_{12}(x_0,y_0).$$

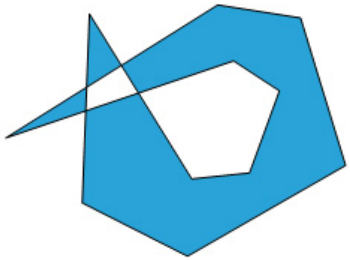
β und γ werden analog berechnet.



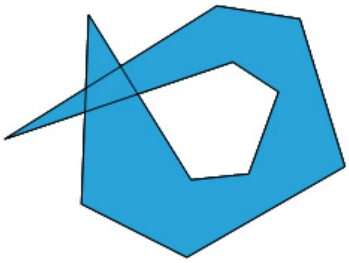
Um das doppelte Zeichnen der Kanten aneinandergrenzender Dreiecke zu vermeiden, werden nur Pixel gezeichnet, deren Mittelpunkt innerhalb eines (exakten) Dreiecks liegen. Pixel genau auf einer Kante sind speziell zu behandeln, z.B. durch Regeln wie „Kanten unten und rechts werden gerendert, Kanten oben und links nicht“. Dadurch stellt man sicher, dass jedes Kantenpixel nur einmal behandelt wird.

Was ist bei einem Polygon innen?

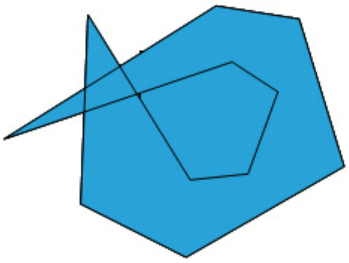
Bevor man mit dem Füllen von Flächen beginnt, muss man sich fragen, was denn zu füllen sei. Bei einer einfachen geschlossenen Kurve ist „innen“ leicht zu definieren, was aber bei komplizierteren Kurven?



so ?

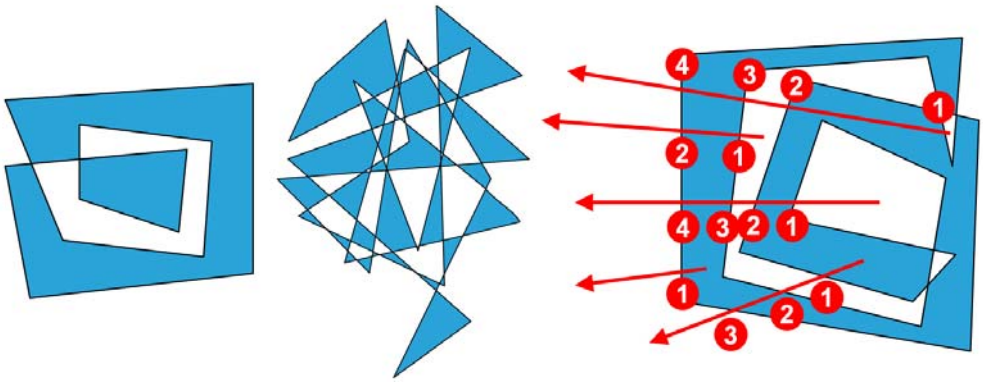


oder so?

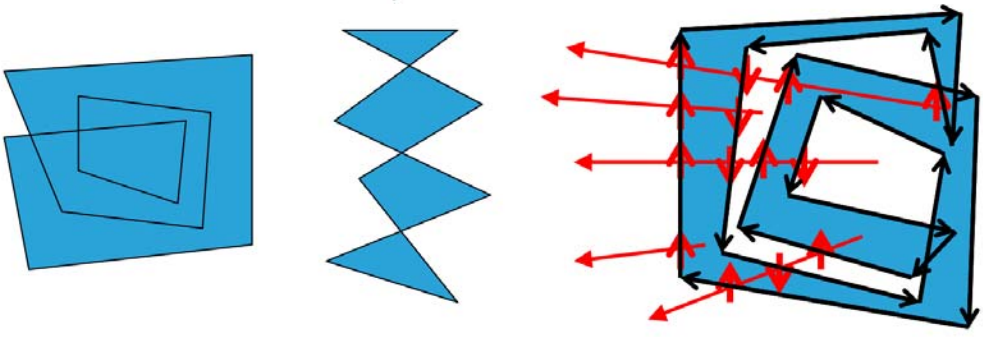


oder so?

Odd-Even-Rule: zieht man von einem Punkt aus einen beliebigen Halbstrahl, so ist der Punkt innerhalb, wenn die Zahl der Schnitte mit der Kurve ungerade ist, ansonsten ist der Punkt außerhalb (in Abb. oben links, sowie alle Bilder rechts). Jede Kante hat also eine Seite innen und die andere außen.



Nonzero-Winding-Number-Rule: Punkte sind außerhalb, wenn sich auf einem beliebigen Halbstrahl gleich viele im Uhrzeigersinn und gegen den Uhrzeigersinn verlaufende Kurvenkanten befinden, ansonsten innerhalb (in Abb. oben Mitte, sowie alle Bilder rechts).



All-In-Rule: alles, was irgendwie umschlossen ist, ist innen. Wird selten verwendet, meist beim Pokern ☺ (in Abb. oben rechts).

Ein Polygon heißt *konvex* wenn alle inneren Winkel kleiner als 180° sind (oberes Bild), andernfalls *konkav* (unteres Bild). Da konvexe Polygone viel weniger Sonderfälle erzeugen, sind viele Algorithmen für konvexe Polygone ausgelegt (oft sogar nur für Dreiecke). Daher braucht man auch Methoden um konkave Polygone in mehrere konvexe Polygone zu zerteilen (oft in Dreiecke).